CS 525: Advanced Database Organization
06: Even more index structures

Boris Glavic

Slides: adapted from a course taught by Hector Garcia-Molina, Stanford InfoLab

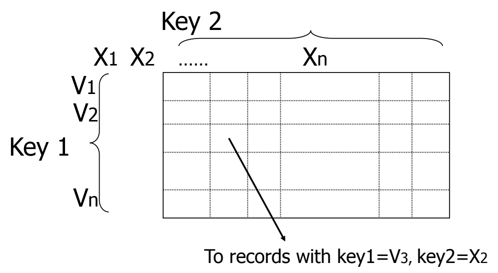
Recap

- We have discussed
  - Conventional Indices
  - B-trees
  - Hashing
  - Trade-offs
  - Multi-key indices
  - Multi-dimensional indices
    - … but no example

Today

- Multi-dimensional index structures
  - kd-Trees (very similar to example before)
  - **Grid File (Grid Index)**
  - Quad Trees
  - **R Trees**
  - **Partitioned Hash**
  - …
- **Bitmap-indices**
- **Tries**

Grid Index



CLAIM

- Can quickly find records with
  - key 1 = V_i ∧ Key 2 = X_j
  - key 1 = V_i
  - key 2 = X_j

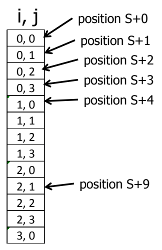
CLAIM

- Can quickly find records with
  - key 1 = V_i ∧ Key 2 = X_j
  - key 1 = V_i
  - key 2 = X_j
- And also ranges….
  - E.g., key 1 ≥ V_i ∧ key 2 < X_j

- How do we find entry i,j in linear structure?

max number of i values N=4

pos(i, j) =



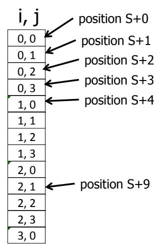
- How do we find entry i,j in linear structure?

max number of i values N=4

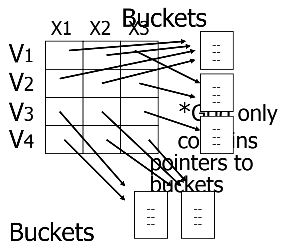
pos(i, j) = S + iN + j

Issue: Cells must be same size, and N must be constant!

Issue: Some cells may overflow, some may be sparse...



Solution: Use Indirection

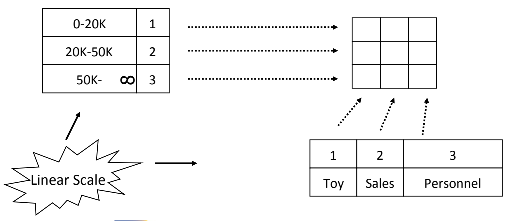


With indirection:

- Grid can be regular without wasting space
- We do have price of indirection

Can also index grid on value ranges

Salary Grid



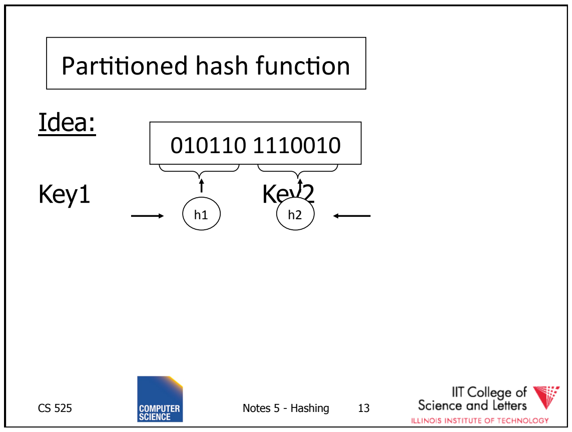
Grid files

⊕ Good for multiple-key search

⊖ Space, management overhead

(nothing is free)

⊖ Need partitioning ranges that evenly split keys



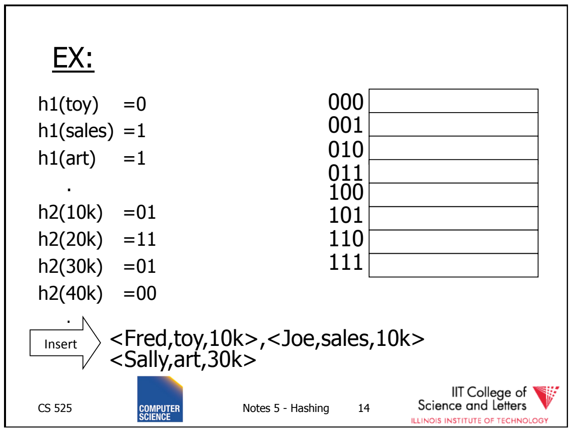
Partitioned hash function

Idea:

010110 1110010

Key1 → h1

Key2 → h2

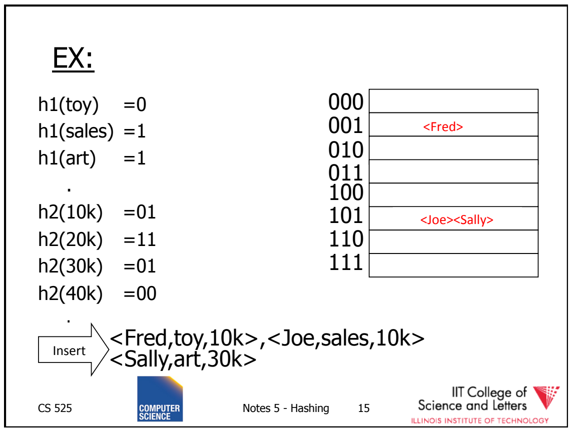


EX:

h1(toy) =0
h1(sales) =1
h1(art) =1
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.

000	
001	
010	
011	
100	
101	
110	
111	

Insert <Fred,toy,10k>,<Joe,sales,10k> <Sally,art,30k>

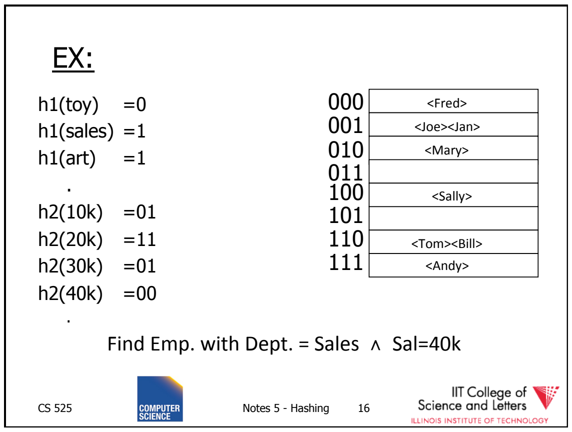


EX:

h1(toy) =0
h1(sales) =1
h1(art) =1
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.

000	
001	<Fred>
010	
011	
100	
101	<Joe><Sally>
110	
111	

Insert <Fred,toy,10k>,<Joe,sales,10k> <Sally,art,30k>

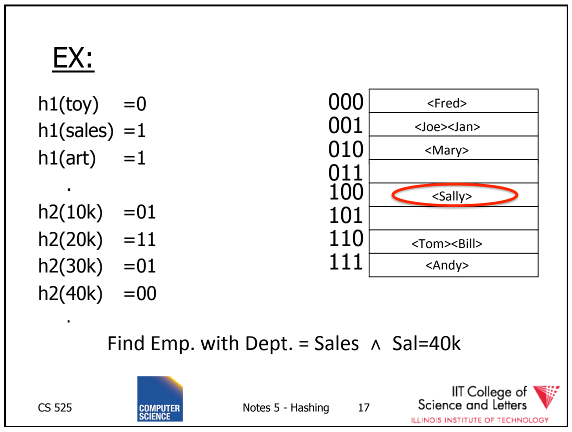


EX:

h1(toy) =0
h1(sales) =1
h1(art) =1
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

Find Emp. with Dept. = Sales ∧ Sal=40k

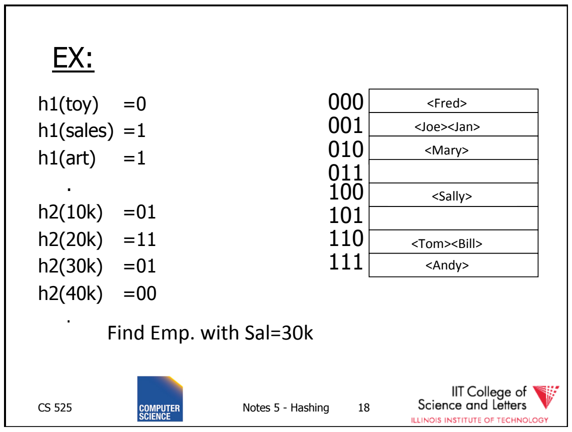


EX:

h1(toy) =0
h1(sales) =1
h1(art) =1
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

Find Emp. with Dept. = Sales ∧ Sal=40k



EX:

h1(toy) =0
h1(sales) =1
h1(art) =1
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

Find Emp. with Sal=30k

EX:

h1(toy) =0
h1(sales) =1
h1(art) =1
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

Find Emp. with Sal=30k

EX:

h1(toy) =0
h1(sales) =1
h1(art) =1
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

Find Emp. with Dept. = Sales

EX:

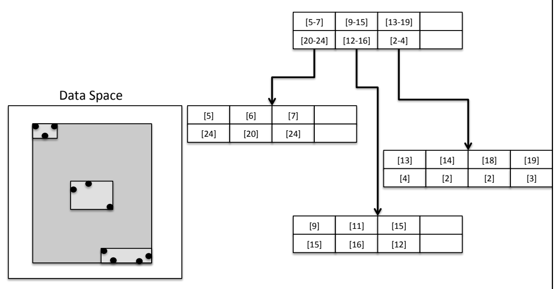
h1(toy) =0
h1(sales) =1
h1(art) =1
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

Find Emp. with Dept. = Sales

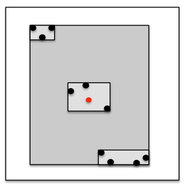
R-tree

- Nodes can store up to **M** entries
  - Minimum fill requirement (depends on variant)
- Each node rectangle in **n**-dimensional space
  - Minimum Bounding Rectangle (MBR) of its children
- MBRs of siblings are allowed to overlap
  - Different from B-trees
- balanced



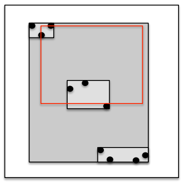
R-tree - Search

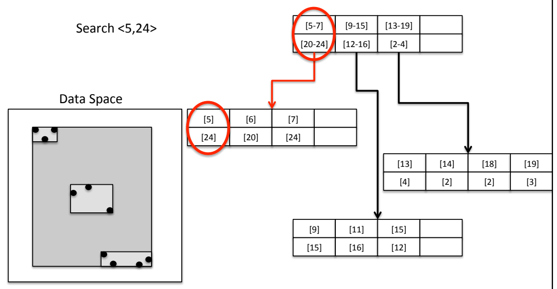
- Point Search
  - Search for p = $<x_i, y_i>$
  - Keep list of potential nodes
    - Needed because of overlap
  - Traverse to child if MBR of child contains p



R-tree - Search

- Point Search
  - Search for points in region = <[x_{min}- x_{max}], [y_{min} -y_{max}]>
  - Keep list of potential nodes
  - Traverse to child if MBR of child overlaps with query region





R-tree - Insert

- Similar to B-tree, but more complex
  - Overlap -> multiple choice where to add entry
  - Split harder because more choice how to split node (compare B-tree = 1 choice)
- 1) Find potential subtrees for current node
  - Choose one for insert (heuristic, e.g., the one the would grow the least)
  - Continue until leaf is found

R-tree - Insert

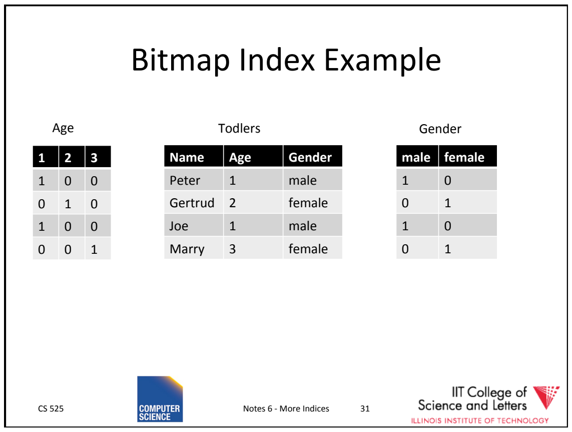
- 2) Insert into leaf
- 3) Leaf is full? -> split
  - Find best split (minimum overlap between new nodes) is hard (O(2^M))
  - Use linear or quadratic heuristics (original paper)
- 4) Adapt parents if necessary

R-tree - Delete

- 1) Find leaf node that contains entry
- 2) Delete entry
- 3) Leaf node underflow?
  - Remove leaf node and cache entries
  - Adapt parents
  - Reinsert deleted entries

Bitmap Index

- Domain of values D = {d_1, …, d_n}
  - Gender {male, female}
  - Age {1, …, 120?}
- Use one vector of bits for each value
  - One bit for each record
    - 0: record has different value in this attribute
    - 1: record has this value



Bitmap Index Example

Age

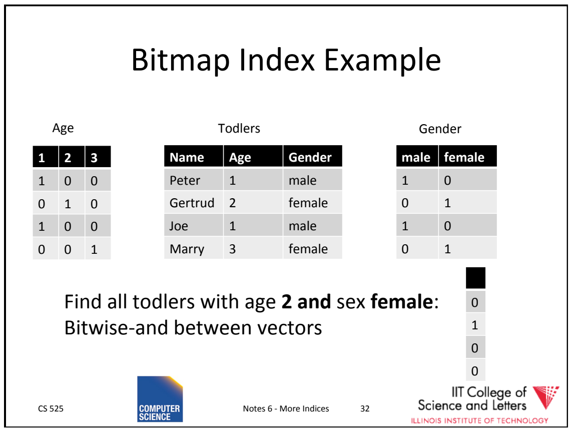
1	2	3
1	0	0
0	1	0
1	0	0
0	0	1

Todlers

Name	Age	Gender
Peter	1	male
Gertrud	2	female
Joe	1	male
Marry	3	female

Gender

male	female
1	0
0	1
1	0
0	1



Bitmap Index Example

Age

1	2	3
1	0	0
0	1	0
1	0	0
0	0	1

Todlers

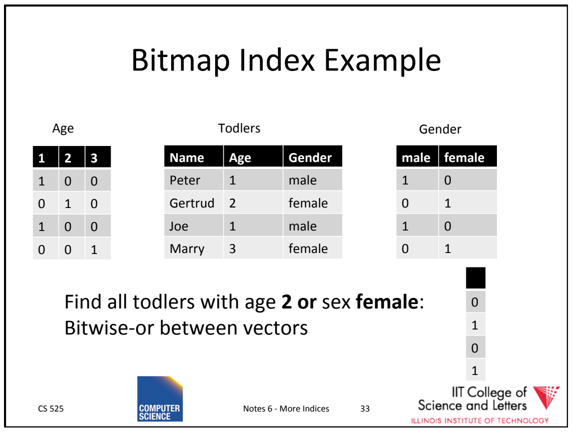
Name	Age	Gender
Peter	1	male
Gertrud	2	female
Joe	1	male
Marry	3	female

Gender

male	female
1	0
0	1
1	0
0	1

Find all todlers with age **2 and** sex **female**:
Bitwise-and between vectors

0
1
0
0



Bitmap Index Example

Age

1	2	3
1	0	0
0	1	0
1	0	0
0	0	1

Todlers

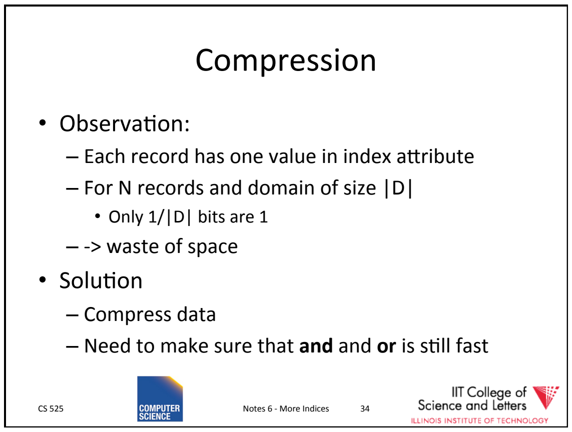
Name	Age	Gender
Peter	1	male
Gertrud	2	female
Joe	1	male
Marry	3	female

Gender

male	female
1	0
0	1
1	0
0	1

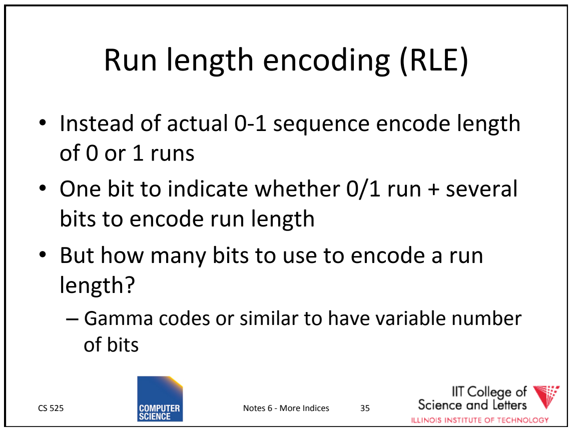
Find all todlers with age **2 or** sex **female**:
Bitwise-or between vectors

0
1
0
1



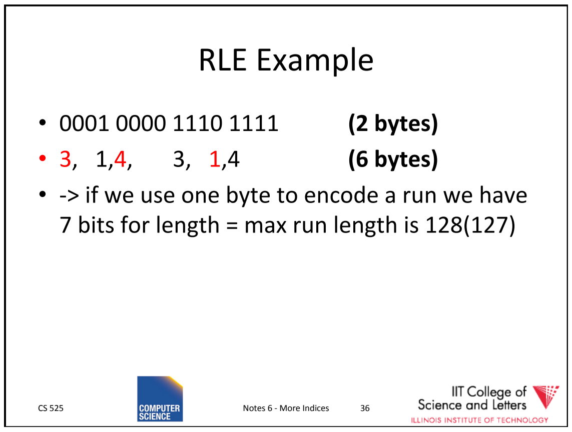
Compression

- Observation:
  - Each record has one value in index attribute
  - For N records and domain of size |D|
    - Only 1/|D| bits are 1
  - -> waste of space
- Solution
  - Compress data
  - Need to make sure that **and** and **or** is still fast



Run length encoding (RLE)

- Instead of actual 0-1 sequence encode length of 0 or 1 runs
- One bit to indicate whether 0/1 run + several bits to encode run length
- But how many bits to use to encode a run length?
  - Gamma codes or similar to have variable number of bits



RLE Example

- 0001 0000 1110 1111 **(2 bytes)**
- 3, 1,4, 3, 1,4 **(6 bytes)**
- -> if we use one byte to encode a run we have 7 bits for length = max run length is 128(127)

Elias Gamma Codes

- X = 2^N + (x mod 2^N)
  – Write N as N zeros followed by one 1
  – Write (x mod 2^N) as N bit number
- 18 = 2^4 + 2 = 000010010

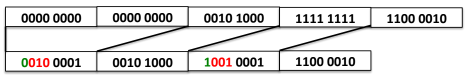
- 0001 0000 1110 1111 **(2 bytes)**
- 3, 1,4, 3, 1,4 **(6 bytes)**
- 0111 0010 0011 1001 00 **(3 bytes)**

Hybrid Encoding

- Run length encoding
  – Can waste space
  – And/or run length not aligned to byte/word boundaries
- Encode some bytes of sequence as is and only store long runs as run length
  – EWAH
  – BBC (that's what Oracle uses)

Extended Word aligned Hybrid (EWAH)

- Segment sequence in machine words (64bit)
- Use two types of words to encode
  – Literal words, taken directly from input sequence
  – Run words
    - ½ word is used to encode a run
    - ½ word is used to encode how many literals follow



Bitmap Indices

- Fast for read intensive workloads
  – Used a lot in datawarehousing
- Often build on the fly during query processing
  – As we will see later in class

Trie

- From Retrieval
- Tree index structure
- Keys are sequences of values from a domain D
  – D = {0,1}
  – D = {a,b,c,….,z}
- Key size may or may not be fixed
  – Store 4-byte integers using D = {0,1} (32 elements)
  – Strings using D={a,…,z} (arbitrary length)

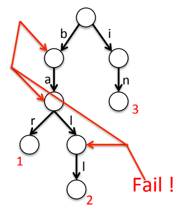
Trie

- Each node has pointers to |D| child nodes
  – One for each value of D
- Searching for a key k = [d_1, …, d_n]
  – Start at the root
  – Follow child for value d_i

Trie Example

Words: bar, ball, in

Search for **bald**



Tries Implementation

- 1) Each node has an array of child pointers
- 2) Each node has a list of hash table of child pointers
- 3) array compression schemes derived from compressed DFA representations

Summary

Discussion:
- Conventional Indices
- B-trees
- Hashing (extensible, linear)
- SQL Index Definition
- Index vs. Hash
- Multiple Key Access
  - Multi Dimensional Indices
    Variations: Grid, R-tree,
  - Partitioned Hash
- Bitmap indices and compression
- Tries